UserGuide

August 3, 2024

[1]:
```
# ---------------- For Developers Only␣
 ↪-------------------------------------------------
# ---------------- Check current working directory for the notebooks␣
 ↪--------------------
import os

# Move one directory up
os.chdir("..")
print("Current Directory:", os.getcwd())
```

```
Current Directory: G:\github-aditya0by0\stream-viz\stream_viz
```

[2]:
```
# ----------------- Use this to move up to the ROOT directory of the project␣
 ↪-----------
# ---------------- For Developers Only␣
 ↪-------------------------------------------------
# ---------------- Move one directory up from the current working directory ␣
 ↪-----------
from pathlib import Path
import os

# Move one directory up
current_dir = Path.cwd()
parent_dir = current_dir.parent
os.chdir(parent_dir)
print("Current Directory:", Path.cwd())
```

```
Current Directory: G:\github-aditya0by0\stream-viz
```

0.1 Data Encoder Implementations

Our library includes three distinct implementations for data encoders, each tailored for specific types of data:

1. `NormalDataEncoder`:
   - Designed for normal data, such as *cfpdss*, without any missing values.
2. `MissingDataEncoder`:
   - Suitable for normal data, like *cfpdss*, that may contain missing values.

3. `KappaStrategyDataEncoder`:
    - Tailored for strategy data, such as *experiments*, without any missing values.

Inheritance and Common Methods All the above implementations inherit from the `DataEncoder` abstract class, ensuring they adhere to a common interface. The abstract class enforces the implementation of the following methods:

- `read_csv_data(filepath_or_buffer, *args, **kwargs)`:
  - Reads CSV data from the specified file path and stores it in the `original_data` class attribute.
  - Internally utilizes the pandas `read_csv` method.
- `encode_data()`:
  - Encodes the data according to the defined process and stores the result in the `encoded_data` class attribute.

Special Considerations for Cfpdss Data For `cfpdss`-related data encoders, there is an additional contract to handle the separation of the target variable from the rest of the data. Thus, the encoded data is stored in separate class attributes:

- `X_encoded_data`: Stores the features (input data).
- `y_encoded_data`: Stores the target variable.

Therefore, when working with `cfpdss` data encoders, use `X_encoded_data` and `y_encoded_data` instead of the `encoded_data` attribute mentioned above.

[3]:
```
from stream_viz.data_encoders.cfpdss_data_encoder import NormalDataEncoder
from stream_viz.utils.constants import _NORMAL_DATA_PATH  # Variable only for␣
↪developers

normal_encoder = NormalDataEncoder()
# Here, add path to your file, the below variable is for internal use only.
# Add relevant/neccessary parameters supported by pandas.read_csv, if required
normal_encoder.read_csv_data(filepath_or_buffer=_NORMAL_DATA_PATH)
normal_encoder.encode_data()
normal_encoder.X_encoded_data.head()
```

[3]:

	c5_b	c6_b	c7_b	c8_b	c9_b	n0	n1	n2	n3	n4
0	0	0	1	0	0	0.528245	0.598345	0.558432	0.482846	0.612024
1	0	0	0	1	1	0.662432	0.423329	0.487623	0.454495	0.452664
2	0	0	0	1	1	0.562990	0.576429	0.545916	0.370166	0.543403
3	0	0	0	1	1	0.475311	0.566046	0.539992	0.421434	0.544852
4	1	0	0	1	0	0.370579	0.554642	0.536804	0.223743	0.392332

[5]:
```
from stream_viz.data_encoders.cfpdss_data_encoder import MissingDataEncoder
from stream_viz.utils.constants import (
    _MISSING_DATA_PATH,
)  # Variable only for developers

missing_encoder = MissingDataEncoder()
missing_encoder.read_csv_data(
    filepath_or_buffer=_MISSING_DATA_PATH,  # Here, add path to your file, this␣
 ↪variable is for internal use only.
    index_col=[
        0
    ],  # Add relevant/neccessary parameters supported by pandas.read_csv, if␣
 ↪required
)
missing_encoder.encode_data()
missing_encoder.X_encoded_data.head()
```

[5]:

	c5_b	c6_b	c7_b	c8_b	c9_b	n0	n1	n2	n3	n4
0	0.0	0.0	1.0	0.0	0.0	0.530356	0.598345	0.519161	0.478557	0.620371
1	0.0	0.0	0.0	1.0	1.0	0.672618	0.423329	0.442055	0.449888	0.458838
2	0.0	0.0	0.0	1.0	1.0	0.567192	0.576429	0.505532	0.364614	0.550814
3	0.0	0.0	0.0	1.0	1.0	0.474236	0.566046	0.499081	0.416457	0.552283
4	1.0	0.0	0.0	1.0	0.0	0.363202	0.554642	0.495610	0.216550	0.397683

[7]:
```
from stream_viz.data_encoders.strategy_data_encoder import␣
 ↪KappaStrategyDataEncoder
from stream_viz.utils.constants import (
    _LEARNING_STRATEGY_DATA_PATH,
)  # Variable only for developers

kappa_encoder = KappaStrategyDataEncoder()
kappa_encoder.read_csv_data(
    filepath_or_buffer=_LEARNING_STRATEGY_DATA_PATH,  # Here, add path to your␣
 ↪file, this variable is for internal use only.
    header=[
        0,
        1,
        2,
    ],  # Add relevant/neccessary parameters supported by pandas.read_csv, if␣
 ↪required
```

```
    index_col=[0, 1],
)
kappa_encoder.encode_data()
kappa_encoder.encoded_data.head()
```

[7]:

Batch_Start	model_all	model_optimal	model_label	model_feat	model_nafa	\
50	0.593128	0.593128	0.432892	0.593128	0.432892	
100	0.447950	0.409449	0.294671	0.332810	0.296875	
150	0.838710	0.919614	0.388254	0.676375	0.384236	
200	0.880000	0.840000	0.720000	0.760000	0.360000	
250	0.918831	0.959612	0.720000	0.708819	0.295775	

Batch_Start	model_smraed_catc	model_smraed_sumc	model_smraed_prioc	\
50	0.257426	0.257426	0.432892	
100	0.334898	0.296875	0.221184	
150	0.592834	0.634146	0.592834	
200	0.680000	0.680000	0.680000	
250	0.672131	0.708819	0.672131	

Batch_Start	model_smraed_
50	0.593128
100	0.334898
150	0.426230
200	0.520000
250	0.573379

0.2 Plotter Contract

All implementations that involve plotting graphs or figures must adhere to the `Plotter` contract. This contract enforces the use of a `plot` method to generate the figure according to the class's specific logic.

Key Benefits

- **Consistency**: Standardizing the method name across all plotting classes ensures that users don't need to remember different method names for various implementations. Each subclass must implement the `plot` method, making the API predictable and straightforward.
- **Ease of Use**: Users can invoke the plot method with a consistent syntax: `'your_class_object'.plot(*args, **kwargs)`. This uniformity simplifies the process of generating plots and enhances the overall user experience.
- **Maintainability**: Adhering to a common interface makes it easier to maintain and extend the library. All plotting classes follow the same method signature, streamlining both development and debugging.

Example Usage Here's an example demonstrating how to use the `plot` method with an instance of a class implementing the Plotter“‘python:

“‘python # Assuming ‘YourClass’ is a subclass implementing the Plotter contract your_class_instance = YourClass() your_class_instanck

“‘yword_arg=value)

0.3 Drift Detection

For effective drift detection, we utilize implementations of two abstract classes:

1. `DriftDetector(Plotter)`:
   - `update`: This method is called at each stream time point from a `Streamer` implementation. It must update the drift mechanism with a new data point at each call. This is essential for keeping the detector current with incoming data.
   - `detect_drift`: This abstract method should implement the core logic of the drift detection mechanism. Different detection techniques will have their unique implementations of this method.
   - `plot`: Plots the relevant figure for the drifts of the implementation. Visualization is key to understanding and analyzing detected drifts.
2. `Streamer`:
   - **Constructor**: Takes the drift detection mechanism (following the above contract) as input. This ensures that the streamer is linked with the appropriate drift detection logic.
   - `stream_data(X_dataframe, y_df)`: This method takes the whole dataset as input and starts streaming the data. During streaming, it sequentially processes the data points.
   - **Streaming Logic**: Within this logic, the streamer calls the `update` method of the `DriftDetector` contract, passing the relevant data at each stream time point. This interaction is crucial for real-time drift detection.

0.3.1 Key Classes

- **RealConceptDriftDetector**: This class implements the `DriftDetector` for detecting real concept drifts. It leverages specific algorithms tailored to identifying changes in the conditional probability distribution (p(y|X)).
- **DataStreamer**: This class implements the `Streamer` and handles the streaming of data. It ensures that data is fed into the drift detector in a sequential and timely manner.

0.3.2 Note

- For real concept drift detection, we currently have the `RealConceptDriftDetector` class.
- For streaming the data, we use the `DataStreamer` class.

By following these contracts and using the provided classes, we can systematically detect and visualize drifts in streaming data, allowing for timely interventions and adjustments in our models and systems.

0.4 1. Real Concept Drift

Real concept drift refers to changes in the conditional probability distribution $p(y|X)$, which affects the decision boundaries or the target concept. For example, a user's interest in news articles may shift from dwelling houses to holiday homes over time.

McDiarmid Drift Detection Method (MDDM) The **McDiarmid Drift Detection Method (MDDM)** is a technique used to detect real concept drifts. It applies McDiarmid's inequality and utilizes a sliding window approach to monitor changes in data streams. Key features of MDDM include:

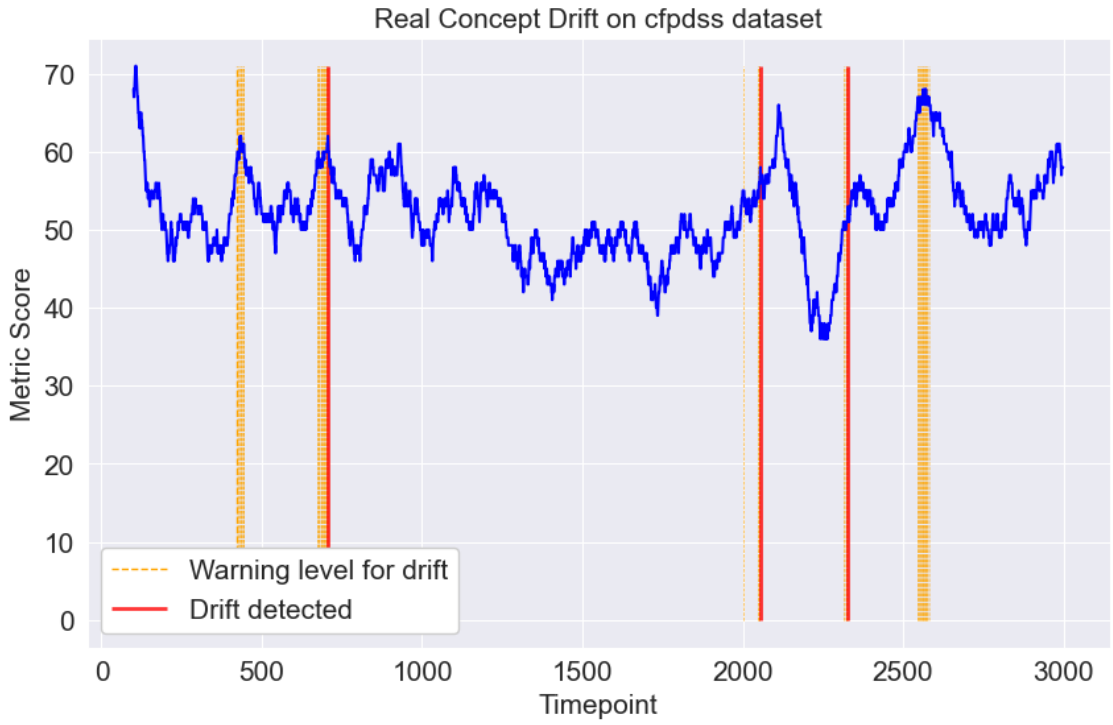
- **Sliding Window Approach**: MDDM processes data in a sliding window to continuously evaluate changes and detect drifts.
- **Weighting Scheme for Elements in the Window**:
  - **Arithmetic Weighting**:
    - Weights are defined as $w_i = 1 + (i + d)$, where $d \geq 0$ is the difference between two consecutive weights. This scheme increases weights linearly with the index of the element.
  - **Geometric Weighting**:
    - Weights are defined as $w_i = r^{(i1)}$, where $r \geq 1$ is the ratio between two consecutive weights. This scheme increases weights exponentially.
  - **Euler Weighting**:
    - Weights are defined as $w_i = r^{(i1)}$ with $r = e^{\lambda}$, where $\lambda \geq 0$. This scheme also increases weights exponentially but uses the base of the natural logarithm.

These weighting schemes help assign different levels of importance to the elements within the slowing sensitive drift detectioni drif

or sensitiv e— For further details, refePesaranghader, A., Viktor, H.L., & Paquet, E. (2017). McDiarmid Drift Detection Methods for Evolving Data Streams. 2018 International Joint Conference on Neural Networks (IJCNN), 1-9.Neural Networks (rg/pdf/1710.02030)

[8]:
```
# ---------------- Real Concept Drift␣
 ↪-------------------------------------------------
from stream_viz.data_streamer import DataStreamer
from stream_viz.real_drift.r_drift_detector import RealConceptDriftDetector

# Initialize DataStreamer with drift detectors
dt_streamer = DataStreamer(
    rcd_detector_obj=RealConceptDriftDetector(),
)

# Stream data and apply drift detection
dt_streamer.stream_data(
    X_df=missing_encoder.X_encoded_data, y_df=missing_encoder.y_encoded_data
)

# Plot results
```

```
dt_streamer.rcd_detector_obj.plot(start_tpt=100, end_tpt=3000)
```



0.5 2. Feature Drift Detection

0.5.1 FeatureDriftDetector Class

The `FeatureDriftDetector` class is designed to monitor and detect feature drift in streaming data using the **Kolmogorov-Smirnov (KS)** test and **Population Stability Index** Test (PSI test). This class can handle numerical features and categorical features, and provides functionality to identify sudden, linear, and gradual drifts in the data. Drift detection is crucial for maintaining the reliability and accuracy of machine learning models, especially in dynamic environments where data distributions can change over time.

Parameters

- **features_list** (`List[str]`): A list of feature names to monitor for drift. It is essential to specify the features of interest to ensure targeted monitoring and efficient detection.
- **window_size** (`int`, optional): The size of the window to use for drift detection. The default value is 300. The window size determines the number of recent data points considered for drift analysis at any given time.
- **ks_test_pval** (`float`, optional): The p-value threshold for the Kolmogorov-Smirnov test. The default value is 0.001. This threshold helps in determining the statistical significance of

the detected drift, ensuring that only meaningful changes are flagged.

- **gap_size** (`int`, optional): The size of the gap between segments when computing gradual drift. The default value is 50. This parameter helps in distinguishing gradual changes from sudden shifts by introducing a gap between the two halves of the data used in the KS test.
- **p_val_threshold** (`float`, optional): The p-value threshold for gradual drift detection. The default value is 0.0001. This threshold is used to determine the significance of gradual changes in the data, providing an additional layer of drift detection sensitivity.
- **psi_threshold** (`float`, optional): The psi threshold for the Population Stability Index. The default value is 0.12. This threshold helps in determining the significance of the detected drift, ensuring that meaningfu changes are detected or flagged.

Drift Detection Methodology Drift detection in the `FeatureDriftDetector` is performed using the Kolmogorov-Smirnov (KS) test for numerical features and Population Stability Index Test (PSI test) for categorical features, which compares the distributions of feature values within a sliding window of data. The following steps outline the drift detection process:

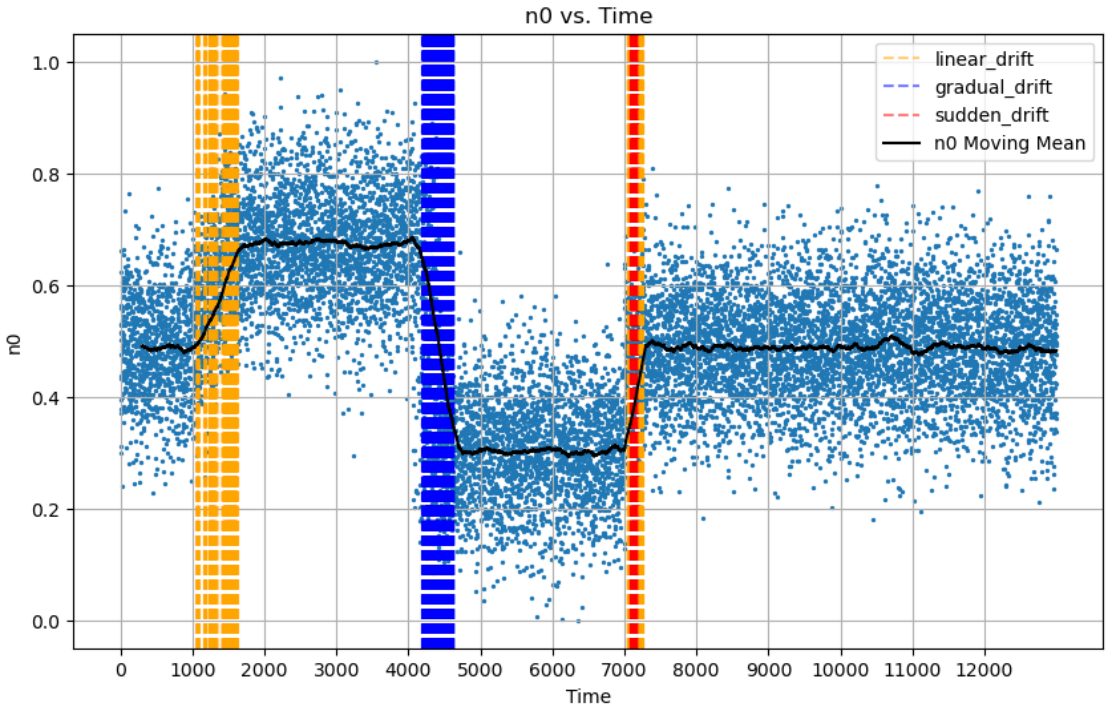
1. **Window Data Segmentation**:
   - The feature values within the current window are split into two halves: the first half and the second half.
   - Additionally, for detecting gradual drift, the data is further segmented into overlapping parts by introducing a gap between segments. Specifically, the first half of the data ends before the gap, and the second half of the data starts after the gap. This method ensures that gradual changes are not masked by immediate changes around the midpoint of the window.
2. **Kolmogorov-Smirnov Test**:
   - The KS test is applied for numerical features to compare the distributions of the first and second halves.
   - If the p-value from the KS test is below the specified threshold (`ks_test_pval`), it indicates a significant difference between the two halves, suggesting potential drift.
3. **Population Stability Index Test**:
   - The PSI test is applied for categorical features to compare the distributions of the first and second halves.
   - If the psi value from the PSI test is above the specified threshold (`psi_threshold`), it indicates a significant difference between two halves, indicating potential drift.
4. **Gradual Drift Detection**:
   - Another KS test and PSI test are conducted on the overlapping parts of the data to detect gradual drift. The overlapping parts are defined by excluding the gap in the middle of the window.
   - If the p-value from this test is below the gradual drift threshold (`p_val_threshold`), it suggests gradual changes in the data distribution.
   - Similarly, if the psi value from this test is above the psi threshold, it indicates gradual drift in data distribution.
5. **Drift Type Identification**:
   - If drift is detected, the type of drift is determined based on the mean difference between the halves:
     - **Sudden Drift**: Identified if the mean difference is significant and exceeds the stan-

dard deviation of the window data.

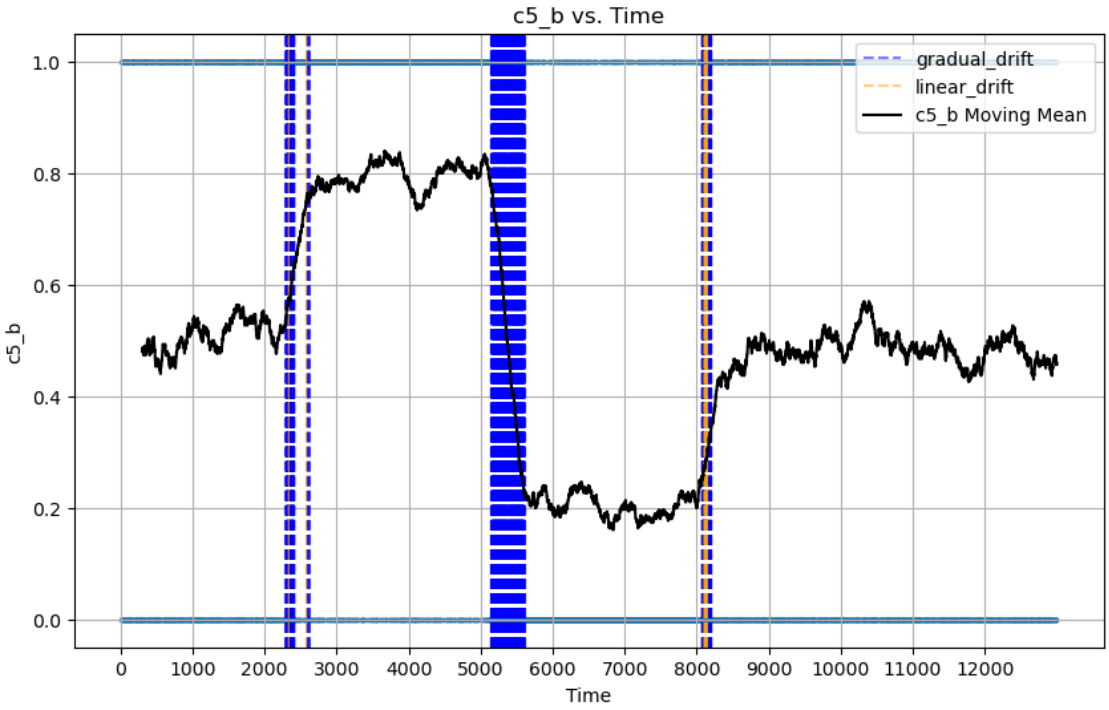
- **Linear Drift**: Identified if the mean difference is positive but does not exceed the standard deviation.
- **Gradual Drift**: Identified if the gradual drift test detects a significant difference.

[5]:
```
from stream_viz.data_streamer import DataStreamer
from stream_viz.feature_drift.f_drift_detector import FeatureDriftDetector

dt_streamer = DataStreamer(
    fd_detector_obj=FeatureDriftDetector(data_encoder=normal_encoder)
)
dt_streamer.stream_data(
    X_df=normal_encoder.X_encoded_data, y_df=normal_encoder.y_encoded_data
)
```

[6]:
```
dt_streamer.fd_detector_obj.plot(feature_name="n0")
```



[6]:
```
dt_streamer.fd_detector_obj.plot(feature_name="c5")
```



0.6 3. Velocity Plots

To analyze velocity, we provide two implementations under the `Velocity(Plotter)` contract:

1. `StackedBarChart`:
   - **Purpose**: Plots a stacked bar chart for a given single **categorical** feature, for each time period.
   - **Details**: Each time period consists of several time points, allowing you to observe changes in the distribution of categorical values over time.
2. `RollingMeanStds`:
   - **Purpose**: Plots rolling/moving means and standard deviations given a window size for **numerical** features.
   - **Details**: Multiple numerical features can be given as input. This plot generates sub-plots for each feature with a common x-axis, facilitating better comparative visual analysis across features.
3. `StreamGraph`:
   - **Purpose**: Plots a stream graph for a given single **categorical** feature, for each time period/ window size.
   - **Details**: Each time period consists of 50 time points, allowing you to observe changes in the distribution of categorical values over time.

0.6.1 Combined Implementation

The above classes are combined into a single class, `FeatureVelocity`, for ease of use, ensuring a unified approach to plotting velocity.

- `FeatureVelocity`:
  - **Purpose**: Simplifies the process by allowing you to call a single class for velocity plots.
  - **Functionality**: This implementation takes the feature name(s) and differentiates between numerical and categorical features based on the metadata of encoders. It then internally calls the respective class object (`StackedBarChart` for categorical features and `RollingMeanStds` for numerical features).

0.6.2 Usage

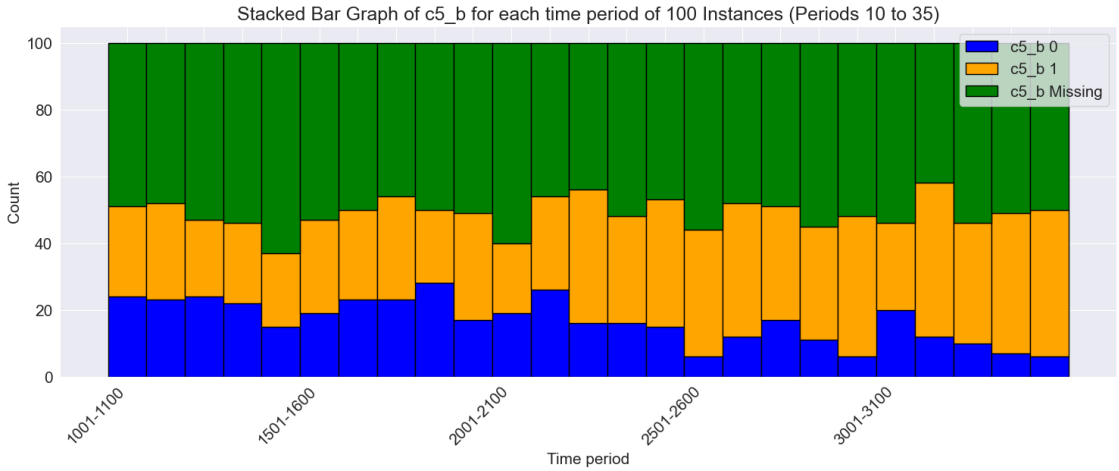
- **Standalone Approach**: Users have the option to use `StackedBarChart` or `RollingMeanStds` individually if they wish to focus on a specific type of feature.
- **Combined Approach**: Alternatively, users can employ the `FeatureVelocity` class to handle both categorical and numerical features in a streamlined manner.
- **Standalone Approach**: Users can the `StreamGraph` class to plot stream graph for categorical features.

0.6.3 Key Features

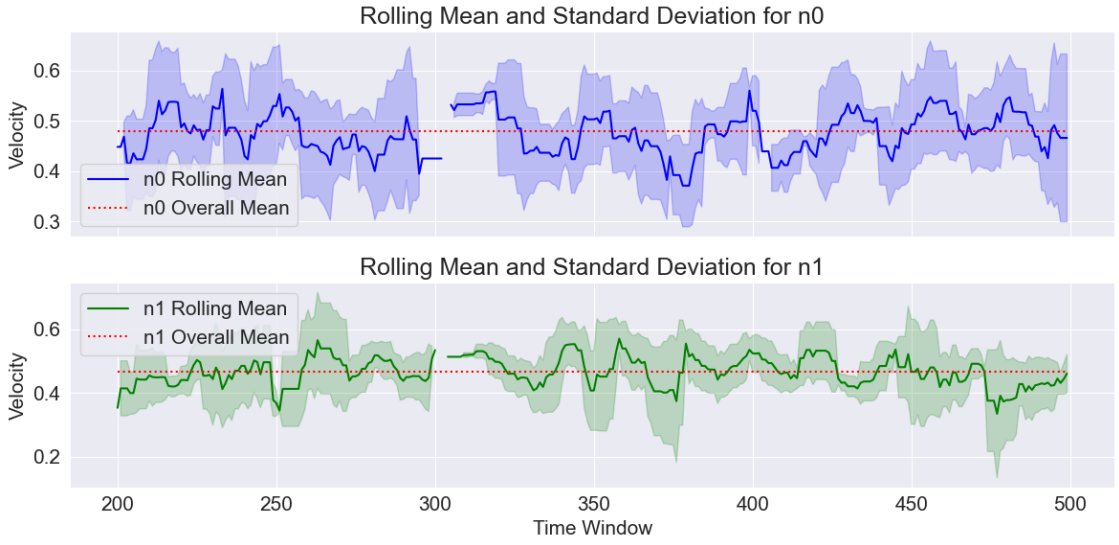
- **Categorical Feature Analysis**: Visualize the distribution changes of categorical features over time with stacked bar charts.
- **Numerical Feature Analysis**: Monitor trends and variations in numerical features using rolling means and standard deviations.
- **Unified Interface**: The `FeatureVelocity` class provides a convenient way to generate both types of plots, enhancing flexibility and ease of use.

By leveraging these implementations, users can perform a comprehensive analysis of feature velocities, gaining valuable insights into how features change over time.

[18]:
```
from stream_viz.velocity.velocity_charts import FeatureVelocity

feature_vel = FeatureVelocity(missing_encoder)
# Plot categorical feature
feature_vel.plot(
    features="c5", chunk_size=100, start_period=10, end_period=35,␣
 ↪x_label_every=5
)
```



[13]:
```
from stream_viz.velocity.velocity_charts import FeatureVelocity

feature_vel = FeatureVelocity(missing_encoder)
# Plott numerical features
feature_vel.plot(features=["n0", "n1"], window_size=10, start_tp=200, end_tp=500)
```

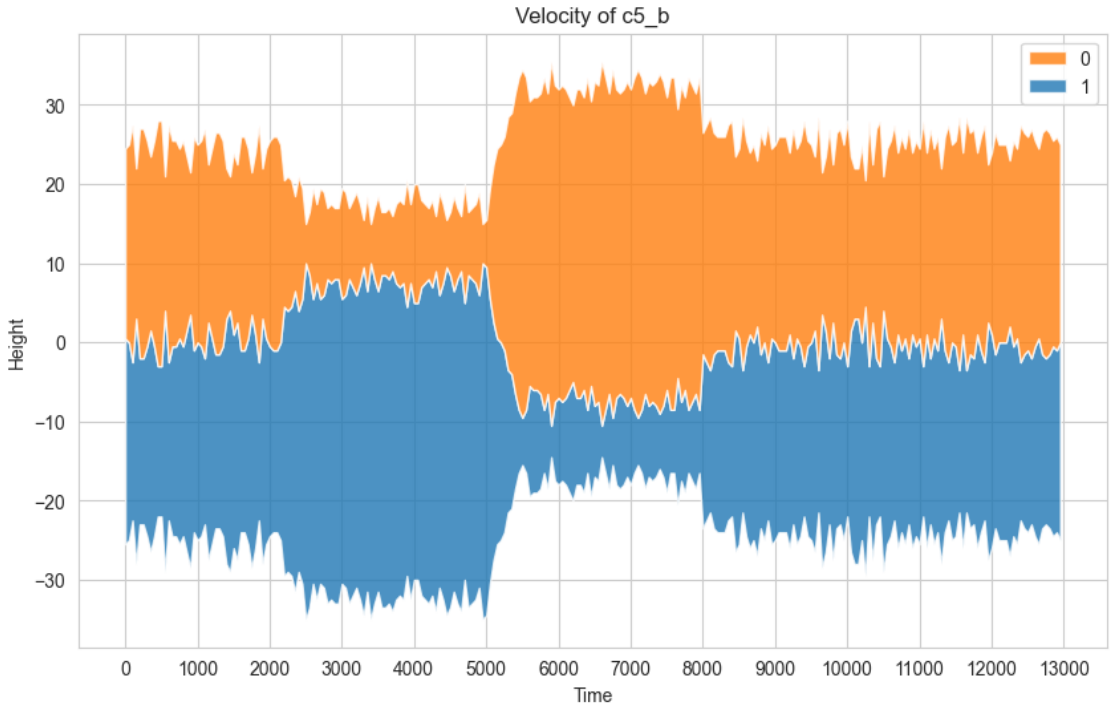


0.6.4 Stream Graphs

- A stream graph shows how categories in a feature change over time, with the vertical axis representing the magnitude of each category at each time point.
- The baseline is not fixed, allowing for negative values on the Y-axis.
- The stream graph is used typically used for categorical data, it can be used for numerical data provided there are bins for the numerical data (This is not implemented in the current

version).

[6]:
```
from stream_viz.velocity.velocity_charts import StreamGraph

streamgraph = StreamGraph(normal_encoder)
streamgraph.plot("c5_b")
```



0.7 4. Missingness Plots

In addition to the `Plotter` contract, we have another abstract class named `InteractivePlot`. By inheriting this contract, you can create user-friendly interactive plots, allowing you to tune the parameters using buttons, sliders, or other UI elements. This enhances the capability for rigorous visual analysis.

0.7.1 Implementation Requirements

- `_add_interactive_plot(*args, **kwargs)` : Any class implementing this contract must define the `_add_interactive_plot` abstract method to add custom inputs to the interactive plot.
- `display(*args, **kwargs)` : You can further customize the interactive plot by overriding the `display` method for specific logic.
- **Feature Display**: By default, the plot displays only the first three features of the dataset, but this can be adjusted through the UI elements.

Additionally, any class implementing the `InteractivePlot` contract must also implement the

`Plotter` class or at least the `plot` method inherited from `Plotter`. This ensures consistency and usability across different types of plots.

0.7.2 Usage Instructions

- **Normal Plot**: Call the `plot` method on your object for a standard plot.
- **Interactive Plot**: Call the `display` method for an interactive plot. This dual functionality lets users choose their preferred visualization method.

0.7.3 Key Features

- **User-Friendly Interactivity**: Enhance plots with buttons, sliders, and other UI elements for detailed visual analysis.
- **Customizable Inputs**: Implement the `_add_interactive_plot` method to tailor inputs to your plot's needs.
- **Default Feature Display**: Initially displays the first three features of the dataset, with UI options to adjust.
- **Dual Plotting Functionality**: Easily switch between the standard `plot` method and the interactive `display` method based on user preference.

This framework provides a comprehensive solution for creating both static and interactive plots, enhancing flexibility and ease of use for various data visualization needs.

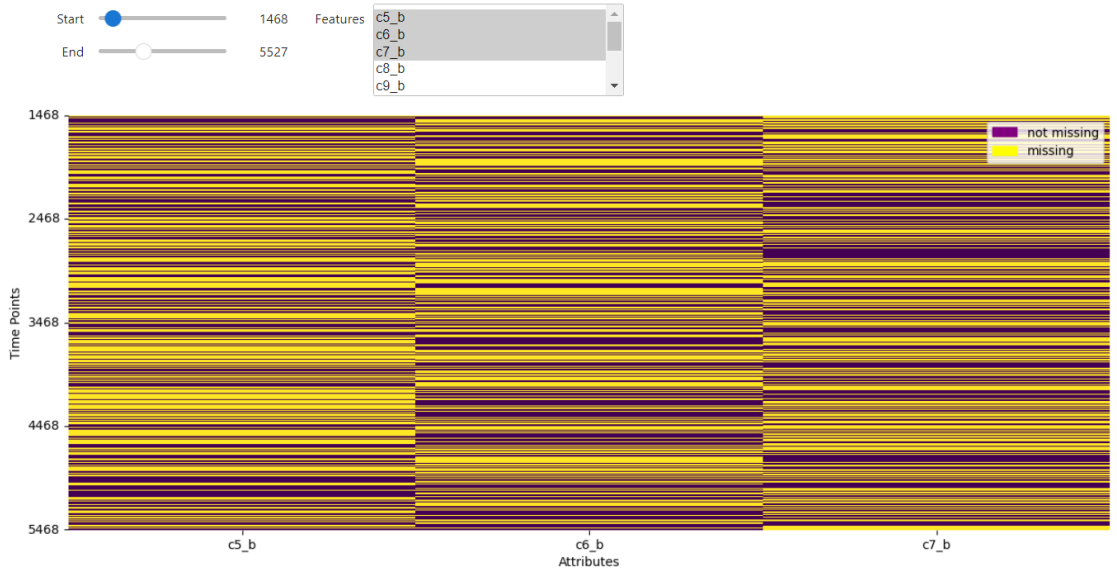
Note: As of now, only `HeatMapPlotter` implements this interface.

[6]:
```
from stream_viz.data_missingness.missingness import HeatmapPlotter

plotter = HeatmapPlotter(missing_encoder.X_encoded_data)
plotter.display()
```

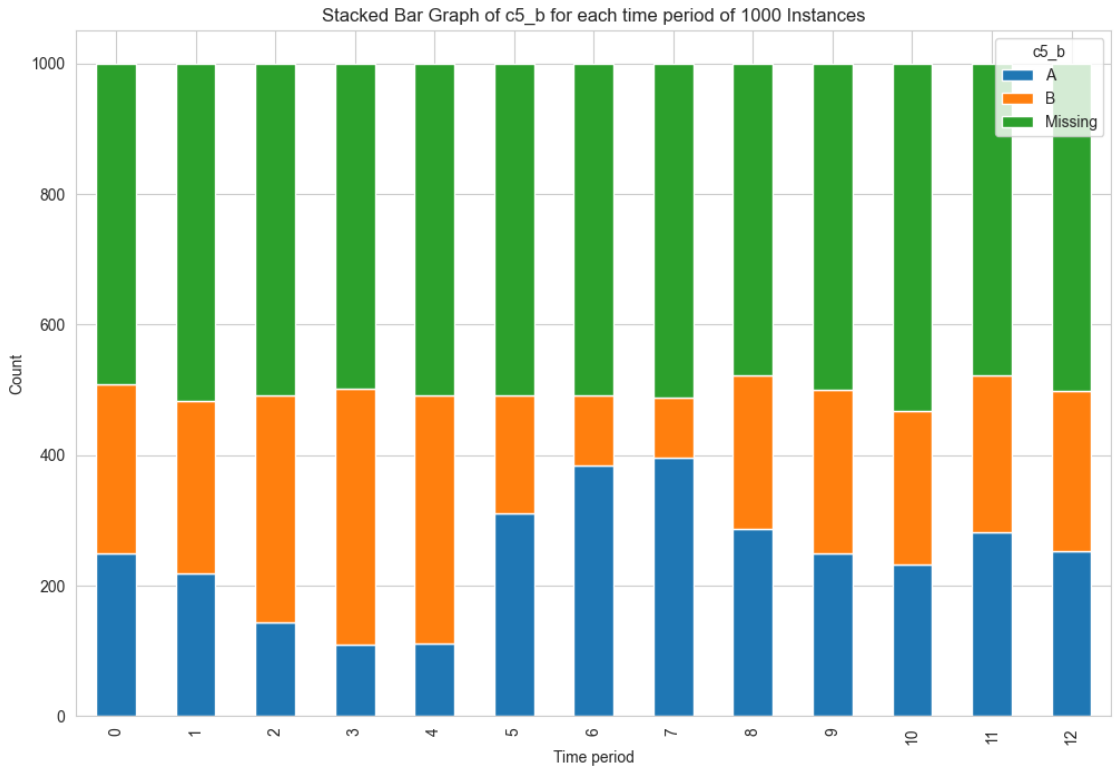
```
HBox(children=(VBox(children=(IntSlider(value=0, description='Start', max=12999),␣
↪IntSlider(value=1000, descri...
```

```
Output()
```

Image of interactive HeatMapPlot below as the interactive plot doesnt't get parsed as static plot for html/view page.



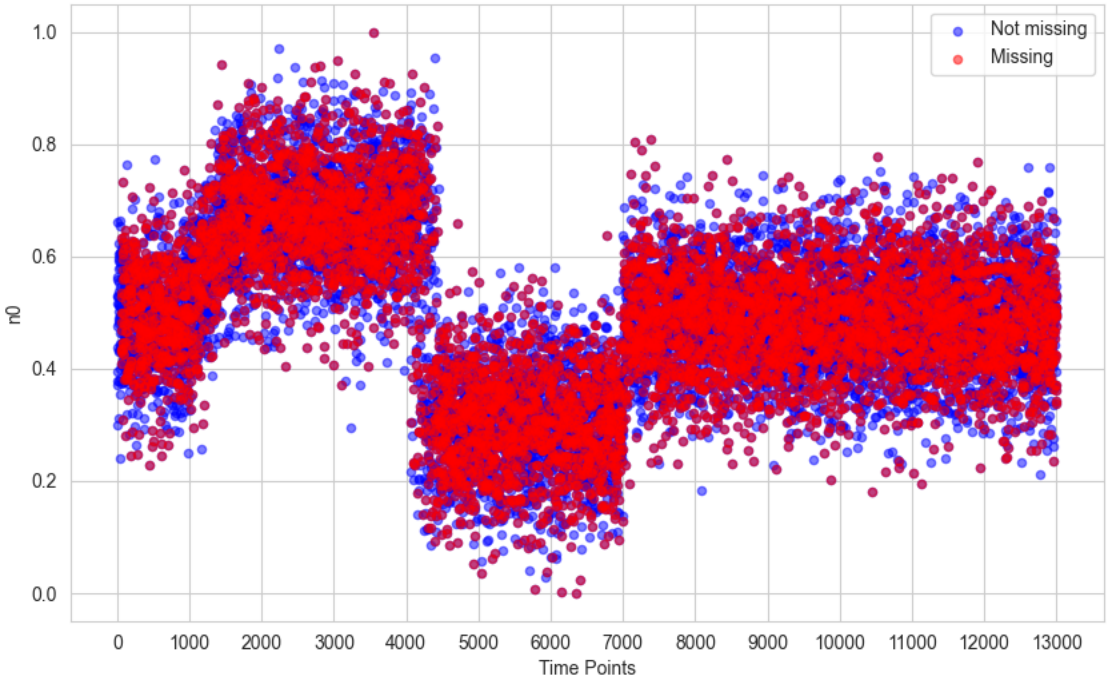
[8]:
```
from stream_viz.data_missingness.missingness import StackedBarGraph

# Create the StackedBarGraph object and plot it
stacked_bar = StackedBarGraph(missing_encoder)
stacked_bar.plot("c5_b", 1000)
```



0.7.4 Scatter Plot

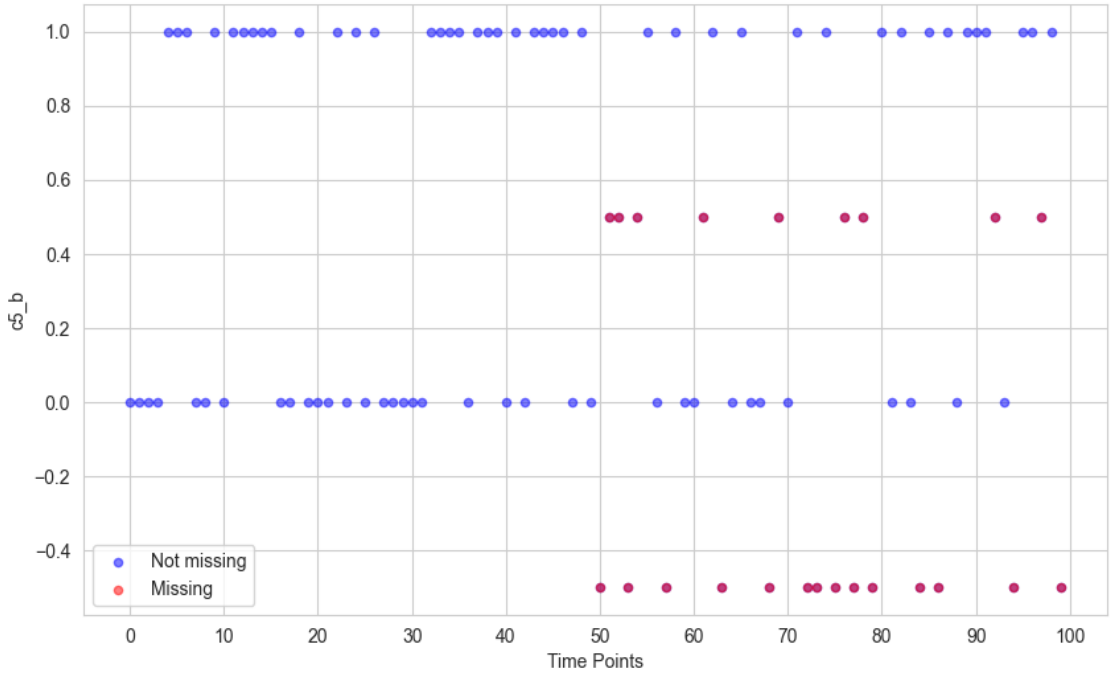
Note: The user should upload both the complete and incomplete datasets to generate this visualization. - The ScatterPlotter class has two methods: plot_numerical and plot_categorical. - The plot_numerical method handles numerical features, displaying all time points on the X-axis. - The plot_categorical method handles categorical features, adjusting missing values to 0.5 after one-hot encoding and limiting the X-axis to the first 100 time points.

[10]:
```
from stream_viz.data_missingness.missingness import ScatterPlotter

scatter_plotter = ScatterPlotter(normal_encoder, missing_encoder)
scatter_plotter.plot_numerical("n0")
```



[11]:
```
scatter_plotter.plot_categorical("c5_b")
```

```
/Users/shreeyacy/GitHub/stream-
viz/stream_viz/data_missingness/missingness.py:310: FutureWarning: Setting an
item of incompatible dtype is deprecated and will raise an error in a future
version of pandas. Value '-0.5' has dtype incompatible with int32, please
explicitly cast to a compatible dtype first.
  normal_df.loc[ind, feature] -= 0.5
```



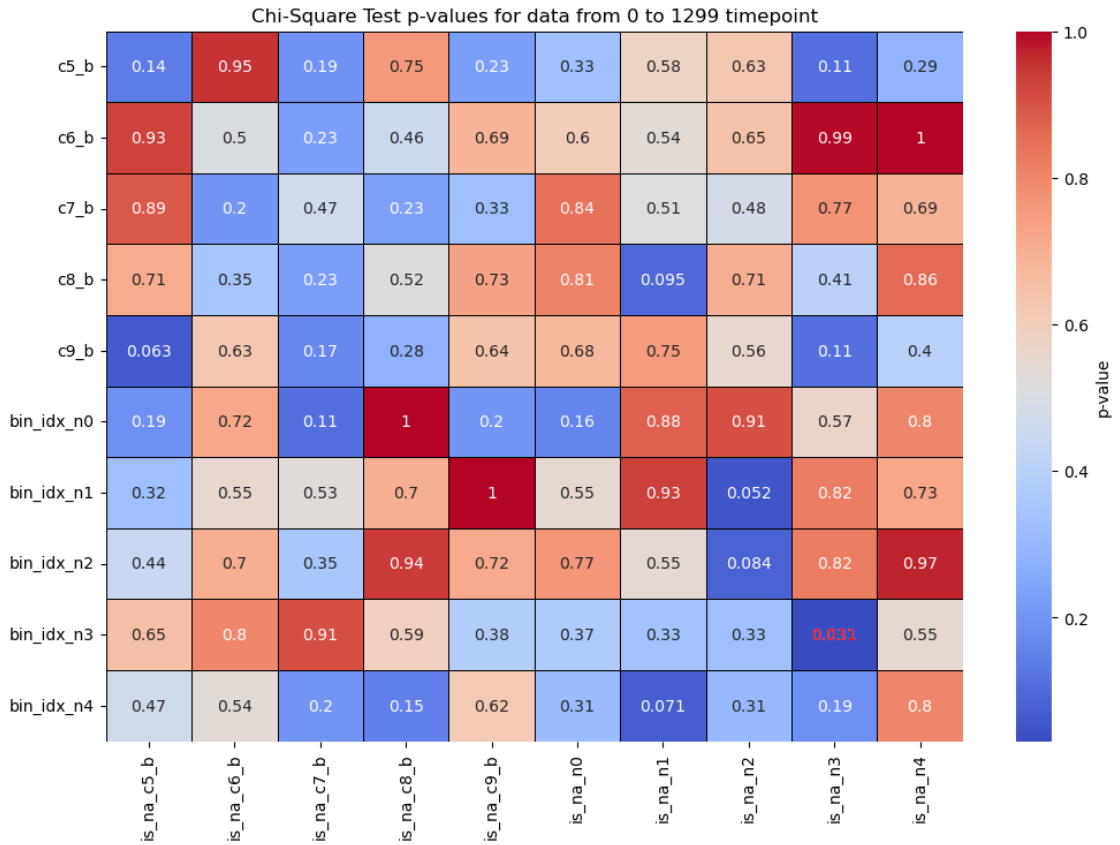
0.7.5 Data Missingness HeatMap Plot (Additional/Optional)

MAR (Missing at Random): - The missing data are related to some of the observed data but not to the missing data itself. - For example, consider two variables: age and income. Income might be missing more frequently for very young people, but the missingness is not due to income itself.

Achieved by: 1. **Binned Numerical Features**: Numerical features are binned with the help of decision trees to categorize them into discrete intervals. 2. **Missing Value Indicators**: Extra columns are added to the dataset to indicate the presence of missing values (is_na_col). 3. **Chi-Square Test**: A Chi-Square test is used to check for dependency between features from non-missing data and the missingness of corresponding features. This test compares the distribution of categorical variables. - **Null Hypothesis**: There is no relationship between the given two variables. - **Significance Level**: Assumed to be 0.05. If the p-value is greater than the significance level, we fail to reject the null hypothesis, indicating that the data is not MAR.

Implemented by: - `MarHeatMap`: This class computes and visualizes the association between categorical columns and missing indicator columns using the chi-square test. You need to provide the missing and non-missing data and call the `plot` method with relevant parameters.

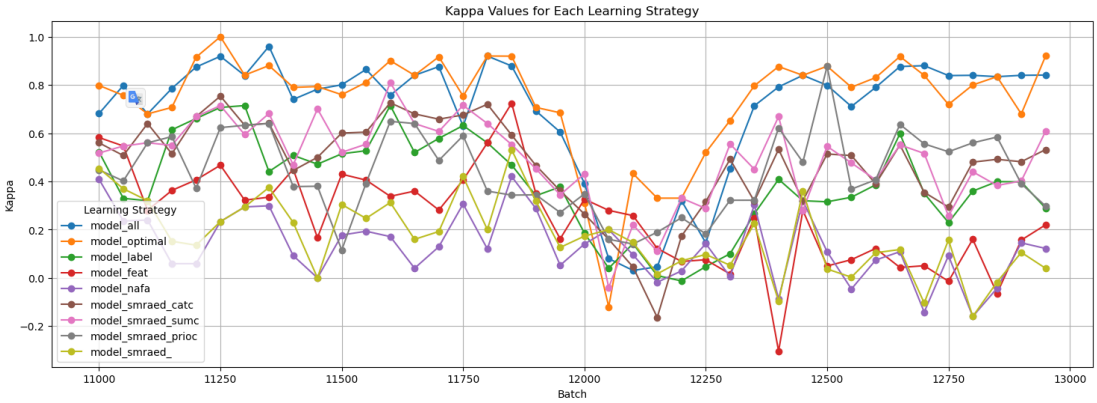
[9]:
```
from stream_viz.data_missingness.missingness import MarHeatMap

mar_hm = MarHeatMap(
    normal_encoder_obj=normal_encoder, missing_encoder_obj=missing_encoder
)
mar_hm.plot(start_tpt=0, end_tpt=1299, significance_level=0.05)
```



0.8 5. Learning Strategies Plot

0.8.1 Strategy Plot

When evaluating various models or strategies on each batch of your dataset, it is common to visualize their performance in a plot. However, when the number of strategies increases, the plot can become cluttered, making it challenging to identify the best-performing strategy at a glance. Below is an example of such a cluttered plot, where the lines representing different strategies overlap, creating visual chaos.



This cluttered visualization requires manual effort to discern which strategy is performing best in each batch, leading to potential oversight of key insights

---

0.8.2 Proposed Strategy Plot

The proposed strategy plot addresses this issue by clearly indicating the winning strategy at each batch. Additionally, it shows the margin or difference by which the winning strategy outperforms the second-best strategy. This enhancement not only simplifies the comparison but also highlights the most effective strategy over time, making it easier to identify the overall winner.

- **Winner Indication**: The plot clearly marks the winning strategy at each batch.
- **Margin Highlight**: It also shows the margin or difference between the winner and the second-best strategy, providing a clear quantitative measure of performance > This refined visualization aids in quickly spotting trends and making informed decisions without the need for manual inspection

[8]:
```
from stream_viz.learning_strategies.strategy_viz import LearningStrategyChart

# Create the learning strategy chart and plot it
LearningStrategyChart(kappa_encoder.encoded_data).plot(start_tpt=11000,␣
↪end_tpt=12950)
```